9.3 Data Types, Abstract and Concrete

So far in this course, we've used the term *data type* to actually mean two different things. Most of the time, we use it to mean a data type in the Python programming language, like `int` or `list` or a data class we've defined. When we use the term "data type" in this way, it is synonymous with the term *Python class*, which is the name the Python language gives to all of its data types. We'll now call refer to these Python classes as **concrete data types**, since they have a concrete implementation in Python code. This is true for both built-in data types, data classes that we define, and the more general classes we learned about in Section 9.2.

However, there's another way we've used the term "data type" that goes all the way back to 1.1 The Different Types of Data: as abstract representations of data that transcend any one specific programming language. For example, the Python `list` class is implemented differently than the Java `ArrayList` or JavaScript `Array`, but all three share some common expectations of what list operations they support. We can describe these common, language-independent list operations by defining an **abstract data type (ADT)**, which defines an entity that stores some kind of data and the operations that can be performed on it. Using the terminology from [Section 9.1], an abstract data type is a pure interface it is concerned only with the *what*—what data is stored, what we can do with this data—and not the *how*—how a computer actually stores this data or implements these operations.

Familiar abstract data types

Let's take a moment here to review some of the collection-based abstract data types we've seen already in this course.[1]

[1] One caveat with this list: while computer scientists generally agree on what the "main" abstract data types are, they often disagree on what operations each one actually supports. You'll notice here that we've taken a fairly conservative approach for specifying operations, limiting ourselves to the most basic ones.

- **Set**
  - Data: a collection of unique elements
  - Operations: get size, insert a value (without introducing duplicates), remove a specified value, check membership in the set.
- **List**
  - Data: an ordered sequence of elements (which may or may not be unique)
  - Operations: get size, access element by index, insert a value at a given index, remove a value at a given index

- **Mapping**
    - Data: a collection of key-value pairs, where each key is unique and associated with a single value
    - Operations: get size, lookup a value for a given key, insert a new key-value pair, remove a key-value pair, update the value associated with a given key
- **Iterable**
    - Data: a collection of values (may or may not be unique)
    - Operations: iterate through the elements of the collection one at a time.

There are a few more foundational abstract data types in computer science that we'll cover in this chapter, and in future courses. We have discussed many of these throughout the semester so far, and have used many in Python. But the true power of ADTs is that they are abstract enough to transcend any individual program or even programming languages. ADTs like lists, sets, and maps form a common vocabulary that is necessary to being a professional computer scientist.

Abstract vs. concrete data types

Abstract data types form a high-level interface between a computer scientist and how the computer stores program data. A concrete data type is an implementation of an abstract data type: unlike abstract data types, they *are* actually concerned with how the data is stored and how their operations are implemented. The creators of the Python programming language took various abstract data types and created a set of built-in concrete data types (classes), making careful decisions about how each class would store its data and implement its methods. Indeed, as Python programmers we benefit from all the work they've put in to create classes that not just support common ADTs, but to make their implementations extremely fast using clever programming techniques.[2]

[2] You'll learn about some of these techniques in CSC263/265!

So a `dict`, for instance, is not itself an abstract data type. But the `dict` data type is an obvious implementation of the Mapping ADT. However, *there is NOT a one-to-one correspondence between abstract data types and concrete data types*, in Python or any other programming language. A single abstract data type can be implemented by many different concrete data types. For example, although the Python `dict` is a natural implementation of the Mapping ADT, we could implement the Mapping ADT instead with a `list`, where each element is a tuple storing a key-value pair:

```python
# A Map using a Python dict
{0: 'hello', 1: 42, 2: 'goodbye'}

# A Map using a Python list
[(0, 'hello'), (1, 42), (2, 'goodbye')]
```

Conversely, every concrete data type can be used to implement multiple ADTs. The Python `list` can be used to implement not just the List ADT, but each of the other above ADTs as well. For instance, think about how you would implement the Set ADT with a `list`, and in particular, how you would avoid duplicates.[3] A `dict` could also implement any of the

[3] Though just because something is possible doesn't mean it is a good idea in practice. Beginning Python programmers often implement use a `list` when all they need is the Set ADT's operations. As we discussed in Section 8.6, this leads to slower programs, and so should be avoided.

ADTs above, and the same is true of the new data structures you will learn in this course.

You might be wondering what is the point of making this distinction—so what if `lists` can implement the Mapping ADT, we'd never use this in "real" Python code when we have a `dict` instead. And that's true! But what this distinction reminds us is that we always have *choices* when implementing an interface. Rather than saying "it's not possible to implement a Map using `list`", we instead say "it is possible to implement a Map using `list`, but this choice is worse than using `dict`".

Any idea why is a `dict` better than `list` at implementing the Mapping ADT? If we ignore the fact that we've been using `dict` for this purpose all along, the answer is not obvious! It comes down to *efficiency*: though `dict` and `list` can both be used to implement the Map ADT, the implementation of `dict` makes the Mapping operations much faster than how we would (straightforwardly) implement the Mapping ADT using a `list`. As we'll see a few times this chapter, running time analysis is one of the key ways to evaluate and compare different implementations of an ADT.

CSC110 Course Notes Home